Parameter Database

Ger van Diepen

10 January 2011

Introduction

BBS uses distributed parameter databases to store the calibration parameters, both instrumental and sky parameters. The `ParmDB` package takes care of maintaining the parameter databases. It can deal with scalar parameters as well as coefficients of parameter functions (funklets) like polynomials.
The main ParmDB interface is tailored towards BBS and written in C++. An interface in python is available for easy and flexible access to parameters, mainly for plotting purposes.
The underlying storage system is formed by the Casacore Tables system as it seems better suited for distributed processing. However, the interface is such that it should be easy to use another storage system like an RDBMS.
The emphasis of ParmDB is on efficiency in storage space and retrieval. Space efficiency is achieved by combining scalar values of an entire solve domain in a single record. Retrieval efficiency is achieved by using an index on parameter name.

Distributed Processing

BBS operates in a distributed way on a distributed MeasurementSet, hence the parameter databases are distributed too. The parameter tables are partitioned in the same way as the MeasurementSet, thus per subband. It means that each subband has its own parameter database tables. A so-called global VDS file describes all these parts to make them globally known.

BBS parameters can be solved per subband or across a group of subbands. In the first case it is clear that each database part has its own unique frequency domains for these parameters. In the latter case information is duplicated in all parts combined in a solution, so a BBS process can always find parameter information in its local database.

Parameter properties

- A parameter name can consist of multiple parts separated by colons. For example,
  ```
  gain:xx:CS001
  ```
  BBS defines the names of all parameters in a standard way.
- A parameter can have multiple values, each one valid for a given frequency/time domain. The value can be a scalar value that is constant for that domain. It can, however, also be a 2-dimensional funklet in frequency and time, for example a polynomial. In that case the value is a 2-dimensional array holding the coefficients of the funklet. The funklet and its coefficients can be scaled to the domain for better numerical behaviour.
- Optionally errors can be attached to the values telling how well a solve behaved.
- It is possible to define a funklet mask telling which funklet coefficients are to be used. By default the higher order coefficients, thus coefficients [i,j] where `(i+j)>=max(nx,ny)`, are not used.
- If a parameter is defined as a funklet, the grade of the funklet must be the same for all domains of that parameter.

- Once a parameter has been solved for, it can be solved again but only on the same solve grid.
- Because BBS uses numerical differentiation, each parameter has a perturbation value that can be used in a relative or absolute way. Relative means that the absolute perturbation value is the perturbation times the parameter coefficient.
  If a solve for multiple domains is done, the absolute perturbation is calculated from the first domain to achieve that the same perturbation is used for the entire solve.
- Default parameter values and attributes are available for parameters that have not been solved yet. In that case the parameter is looked up in the table with default values. If not found, the last part of the name is removed and looked up again. In that way a single default for, say, `gain` can serve as the default for all parameters starting with `gain:`.

Parameter Database Tables

The database consists of three tables:

1. The main table contains the values per parameter per domain.
   The main table also contains some keywords, notably `DefaultFreqStep` and `DefaultTimeStep` defining the default step sizes used by the python interface. These keywords are initialized with the MeasurementSet resolution.
2. The Name table maps a parameter name to a unique name ID. Furthermore it defines the parameter funklet type and some other attributes not dependent on domain.
   It is defined as subtable NAMES of the main table.
3. The DefaultValues table defines the default value and some other default attributes of a parameter.
   It is defined as subtable DEFAULTVALUES of the main table.

Main Table		
Column	***Data Type***	***Description***
NAMEID	int	Parameter name ID (rownr in the Name table).
STARTX	double	Domain start value for the X axis (frequency).
ENDX	double	Domain end value for the X axis (frequency).
STARTY	double	Domain start for the Y axis (time).
ENDY	double	Domain end for the Y axis (time).
INTERVALSX	double[2,nx]	If given and not empty, center and width of each point on the X axis. Otherwise the axis is regularly spaced. Only used if VALUES contains scalar values.
INTERVALSY	double[2,ny]	Same for Y axis.
VALUES	double[nx,ny]	Coefficients if parameter is a funklet (with shape [nx,ny]). Otherwise scalar values of nx*ny points in domain.
ERRORS	double[nx,ny]	If given, the errors attached to VALUES.

NAMES subtable		
Column	***Data Type***	***Description***
NAME	string	Parameter name.
FUNKLETTYPE	int	Defined in `ParmDB/ParmValue.h`
PERTURBATION	double	Perturbation to use for each coefficient when calculating derivatives numerically.
PERT_REL	double	True = perturbation is relative, otherwise absolute.
SOLVABLE	bool[nx,ny]	Optional mask telling which funklet coefficients to use when solving the parameter.
NX	int	Number of funklet coefficients in X direction.
NY	int	Number of funklet coefficients in Y direction.

The name ID (used in the main table) is defined as the row number in the Name table. It means that parameters cannot be deleted from the Name table.

DEFAULTVALUES subtable		
Column	***Data Type***	***Description***
NAME	string	Parameter name or part of it.
FUNKLETTYPE	int	Defined in `ParmDB/ParmValue.h`
PERTURBATION	double	Perturbation to use for each coefficient when calculating derivatives numerically.
PERT_REL	double	True = perturbation is relative, otherwise absolute.
SOLVABLE	bool[nx,ny]	Optional mask telling which funklet coefficients to use when solving the parameter.
DOMAIN	double[4]	Optional domain for funklet (as stx, endx, sty, endy).
VALUES	double[nx,ny]	Coefficients if parameter is a funklet (with shape [nx,ny]). For a scalar parameter nx=ny=1.

Source Parameter Table

Source parameters are stored in the parameter database tables described above. Their names consist of two parts. The first part is the source name, while the second part gives the source parameter, for example Ra. The source type determines which source parameters are available. For example, a point source has fewer parameters than an extended source. Apart from sources with a fixed position, the source type can also define moving objects like the sun.
The currently used source parameter names are:

- Ra, Dec	J2000 position in radians
- I, Q, U, V	Fluxes in Jy
- SpectralIndex:i	i-th coefficient of spectral index log polynomial
- Orientation)	
- MajorAxis)	Gaussian source parameters
- MinorAxis)	
- PolarizedFraction)	
- PolarizationAngle)	Rotation Measure parameters
- RotationMeasure)	

Other parameters can be defined as needed when new source types are added.

Sources can be grouped in patches, so they can be handled jointly by BBS. Often a patch will contain only one source.

Two tables are added to the parameter database to describe source specific information in a way that the sky parameter database can be used as the Local Sky Model.

1. The Source table defines a source and the patch it belongs to.
   It is defined as subtable SOURCES of the main parameter database table.
2. The Patch table describes a patch.
   It is defined as subtable PATCHES of the Source table.

SOURCES subtable		
Column	***Data Type***	***Description***
SOURCENAME	string	Source name.
PATCHID	uint	Id of the patch containing the source. (rownr in Patch table).
SOURCETYPE	int	Defined in `ParmDB/SourceInfo.h`
REFTYPE	string	Frame reference type (J2000, etc.)
SPINX_NTERMS	int	Degree of spectral index (<0 means no spectral index)
SPINX_REFFREQ	double	Reference frequency for spectral index
USE_ROTMEAS	bool	true = use rotation measure for Q,U
SHAPELET_ISCALE	double	Shapelet scale for Stokes I
SHAPELET_QSCALE	double	Shapelet scale for Stokes Q
SHAPELET_USCALE	double	Shapelet scale for Stokes U
SHAPELET_VSCALE	double	Shapelet scale for Stokes V
SHAPELET_ICOEFF	double[n,n]	Shapelet coefficients for Stokes I
SHAPELET_QCOEFF	double[n,n]	Shapelet coefficients for Stokes Q
SHAPELET_UCOEFF	double[n,n]	Shapelet coefficients for Stokes U
SHAPELET_VCOEFF	double[n,n]	Shapelet coefficients for Stokes V

PATCHES subtable		
Column	***Data Type***	***Description***
PATCHNAME	string	Patch name.
CATEGORY	int	1, 2, or 3 for Cat-1, Cat-2, or Cat-3 sources.
APPARENT_ BRIGHTNESS	double	Apparent brightness of patch. Used for peeling in descending order of brightness.
RA	double	J2000 RA of patch center.
DEC	double	J2000 DEC of patch center.

The patch ID (used in the Source table) is defined as the row number in the Patch table. It means that patches cannot be deleted from the Patch table.

Parameter Interface

The `CEP/ParmDB` package contains C++ classes and functions to access the parameter databases. A brief description of the main classes follows; more detailed information is given in the appendix and in the classes themselves.

The following classes are designed for use by BBS:

1. `ParmDB` opens a parameter database. It is a wrapper for classes that implement the database in a storage specific way. Currently only `ParmDBCasa` is implemented.
2. `ParmSet` contains the names of the parameters used in a BBS step.
3. `ParmCache` contains the values and attributes of the parameters in `ParmSet` for a given frequency/time work domain.
4. `Parm` can be used to get and set parameter coefficients and errors, to define a solve domain and grid, and to evaluate a parameter for a given predict grid.
   It is important to note that the solve domain can be smaller than the work domain defined in `ParmCache`, but not larger. When extending the solve domain, the values of the previous solve domain will be used as start values instead of the default values from the DefaultValues table.

These classes give local access to a parameter database, so they are typically used in each distributed BBSKernel process.
Furthermore some helper classes defining domains and grids are used. These are `Box`, `Axis`, and `Grid`.

The `ParmFacade` class is meant for inspection purposes and offers readonly access to a parameter database. Unlike the classes mentioned above, it can also handle a distributed parameter database when given the name of the global VDS file describing the distributed parameter database. In that case it starts distributed processes (using `startdistproc`) that access each database part locally and combines the data from the distributed processes.
Usually `ParmFacade` will not be used directly, but through its python interface `parmdb`. Values are returned as `numpy` arrays that can be plotted easily using, say, `matplotlib`. The distributed aspect of the system is fully hidden from the user.
The python interface is implemented in package `CEP/pyparmdb`.

The `SourceDB` class is the interface to the source parameter database. It is effectively a wrapper around the `ParmDB` class. It adds functionality to find patches and sources that can be used by BBS to do peeling and to know the types of sources.

Several programs and scripts exist to create, fill, or inspect a parameter database.

1. `parmdb` can be used to create or open a local parameter database, add parameters, values, and default values, update or delete them, retrieve them. It is a command line tool, so will not be used very often.
2. `makesourcedb` can be used to create or open a local source parameter database and append sources and patches to it using an ASCII input file. The format of the input file can be described by means of a format line making it possible to handle many different input formats.
3. `showsourcedb` can be used to view the contents of a source data base.
4. `setupparmdb` and `setupparmdb-part` create a distributed parameter database. A local database can be used as a template that will be copied to all parts. If no template database is given, a default one will be used.
   A global VDS file is created describing all database parts. The global VDS file of a MeasurementSet is used as input to know where all parts have to reside.

5. `setupsourcedb` and `setupsourcedb-part` do the same for the source parameter database. They use `makesourcedb` to fill the database parts from an ASCII input file.
6. `startparmdbdistr`, `parmdbremote-scr`, and `parmdbremote` are used by class `ParmFacadeDistr` to handle a distributed parameter database.

<u>Appendix A Python Interface *parmdb*</u>

`parmdb (dbname)`

opens the given database. The database can be a local one given by the name of the main table. It can also be a distributed one given by the name of its global VDS file. In that case `parmdbremote` processes will be started on nodes having access to the distributed database parts. The processes will be ended when closing the database. For example:

```
import lofar.parmdb
pdb = parmdb(dbname)      # open
pdb = 0                   # close
```

`getRange (parmnamepattern="")`

returns the domain range of the matching parameter names (default all names) as
[startfreq, endfreq, starttime, endtime].
The parmnamepattern can be a pattern like wildcarded file names used in a shell.

`getNames (parmnamepattern="")`

returns the list of matching parameter names (default is all names).

```
getValues (parmnamepattern,
           sfreq, efreq, freqstep, stime, etime, timestep,
           asStartEnd=true)
```

calculates the values for the given parameters on the regular grid defined by frequency and time ranges and steps. A range can be given as start/end (is default) or center/width. A step size 0 means that the default held in the parameter database is used. This is usually the frequency and time resolution in the MeasurementSet.
The values are returned as a dict of parameter names to subdicts. Each subdict contains the fields *values*, *freqs*, *times*, *freqwidths*, and *timewidths*. *Values* is a 2D numpy array containing the parameter value for each grid point. The other fields define the grid axes.
Note that numpy arrays are in C-order, thus have shape [nt,nf].

```
getValues(parmnamepattern, sfreq=-1e30, efreq=1e30,
                           stime=-1e30, etime=1e30,
                           asStartEnd=True)
```

as above using the default step sizes.

```
getValuesGrid (parmnamepattern, sfreq=-1e30, efreq=1e30,
                                stime=-1e30, etime=1e30,
                                asStartEnd=True)
```

as above using the grid as defined in the parameter database. Such a grid is only defined for scalar parameters, so for funklets the default step size is used.

```
getCoeff (parmnamepattern, sfreq=-1e30, efreq=1e30,
                           stime=-1e30, etime=1e30,
                           asStartEnd=True)
```

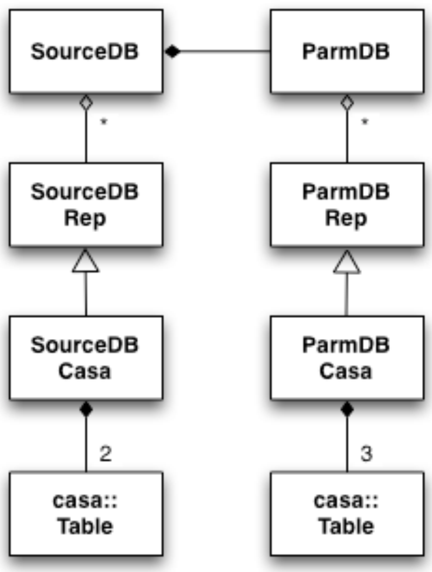
returns the parameter coefficients and errors in a dict as above. Each subdict also contains the field *errors*. Undefined errors are set to -1.
For scalar parameter values the result is effectively the same as for getValuesGrid.
For funklets the *values* and *errors* arrays are 4D arrays with shape [nt,nf,nct,ncf].
[nt,nf] represent the grid and [nct,ncf] the funklet coefficients.

Appendix B Example BBS usage

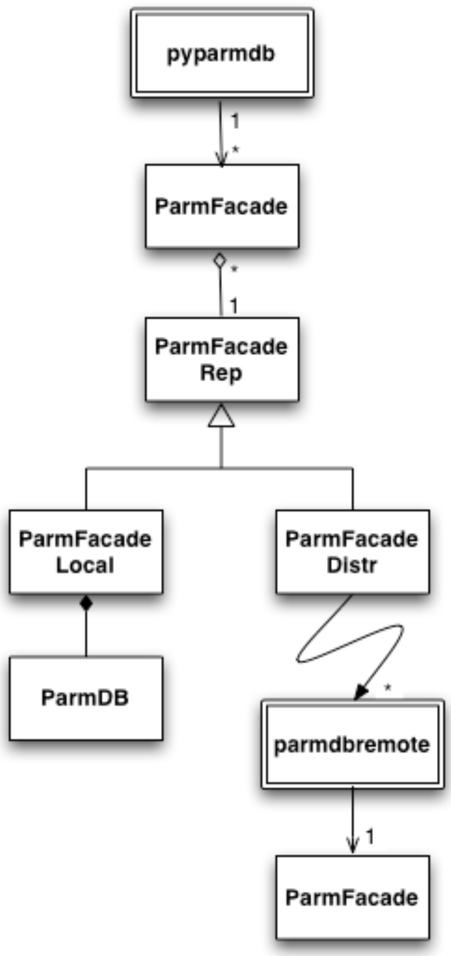
This example is very simple. It uses a few parameters and a solve domain that is as large as the work domain. Yet it shows the basic steps that have to be taken.
Note that the parameters can be scalar or funklet; the code does not need to know.

```
#include <ParmDB/ParmDB.h>
#include <ParmDB/ParmSet.h>
#include <ParmDB/ParmCache.h>
#include <ParmDB/Parm.h>

// Open the instrument and sky ParmDB.
ParmDB skyPdb (ParmDBMeta("sky.pdb"));
ParmDB instrPdb (ParmDBMeta("instr.pdb"));

// Tell which parameters we are interested in.
ParmSet parmSet;
parmSet.add (skyPdb, "src1:RA");
parmSet.add (skyPdb, "src1:DEC");
parmSet.add (instrPdb, "gain:st1");
parmSet.add (instrPdb, "gain:st2");

// Create the ParmCache and the various Parm objects.
ParmCache parmCache(parmSet);
Parm ra1  (parmCache, "src1:RA");
Parm dec1 (parmCache, "src1:DEC");
Parm gain1(parmCache, "gain:st1");
Parm gain2(parmCache, "gain:st2");

// Loop over the work domains.
while (Box domain = nextDomain()) {
  // Set the domain for the cache object.
  parmCache.reset (domain);

  // The gains will be solved, so set their solve domains
  // and get their initial coefficients.
  Grid solveGrid (vector<Box>(1, domain));
  gain1.setSolveGrid (solveGrid);
  gain2.setSolveGrid (solveGrid);
  vector<double> coeffGain1 = gain1.getCoeff (Location(0,0));
  vector<double> coeffGain2 = gain2.getCoeff (Location(0,0));

  // Form the predict grid (10x10 elements).
  Axis::ShPtr ax1(new RegularAxis
   (domain.lowerX(), domain.upperX(), 10, true);
  Axis::ShPtr ax2(new RegularAxis
   (domain.lowerY(), domain.upperY(), 10, true);
  Grid predictGrid (ax1, ax2);

  // Get the parameter values on the predict grid.
  // Also calculate perturbed values for the solvable parms.
  vector<Array<double> > resra, resdec, resgain1, resgain2;
  ra1.getResult  (resra,  predictGrid, false);
  dec1.getResult (resdec, predictGrid, false);
  gain1.getResult (resgain1, predictGrid, true);
  gain2.getResult (resgain2, predictGrid, true);

  // Get the perturbations used.
  vector<double> pertgain1 = gain1.getPerturbations();
  vector<double> pertgain2 = gain2.getPerturbations();

  // Find a new solution for gain1 and gain2 in some way.
  vector<double> solution = findSolution
    (coeffGain1, coeffGain2, pertGain1, pertGain2,
     resra, resdec, resgain1, resgain2);

  // Set the newly found coefficients for gain1 and gain2.
  const double* solPtr = &(solution[0]);
  gain1.setCoeff (Location(0,0), solPtr, coeffGain1.size());
  solPtr += coeffGain1.size();
  gain2.setCoeff (Location(0,0), solPtr, coeffGain2.size());
}
```

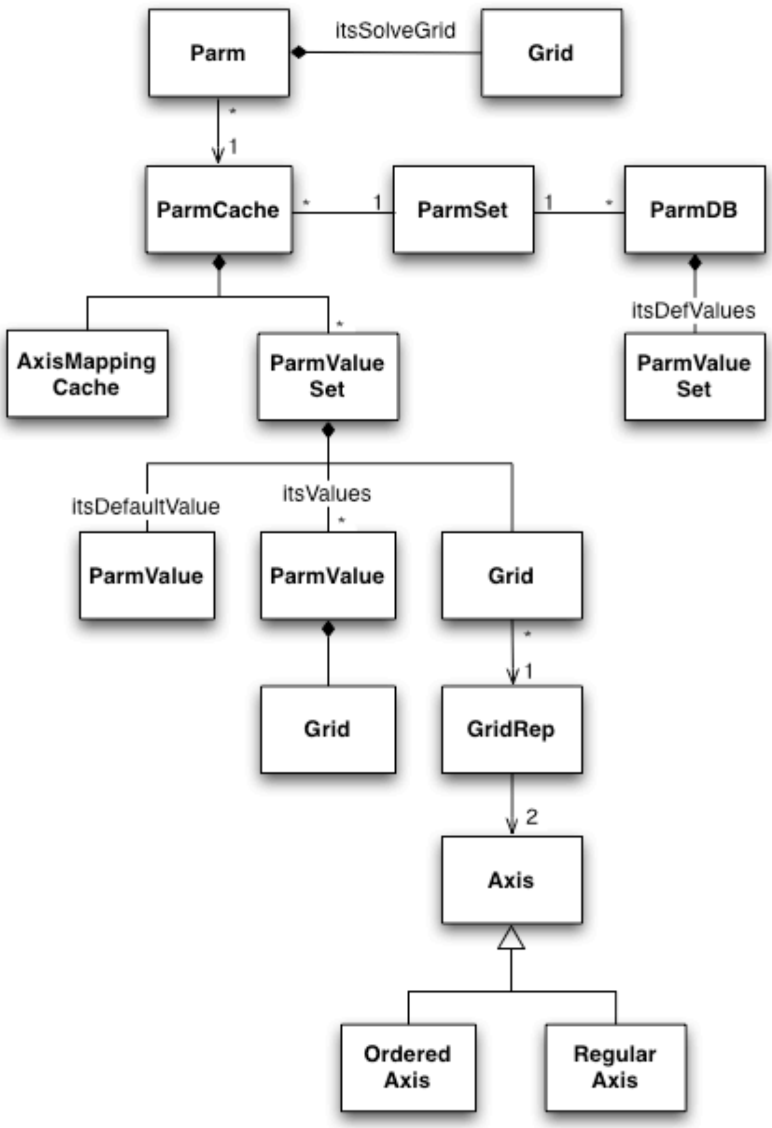
Appendix C Class Diagrams



ParmDB/SourceDB class diagram



ParmFacade class diagram



Parm class diagram